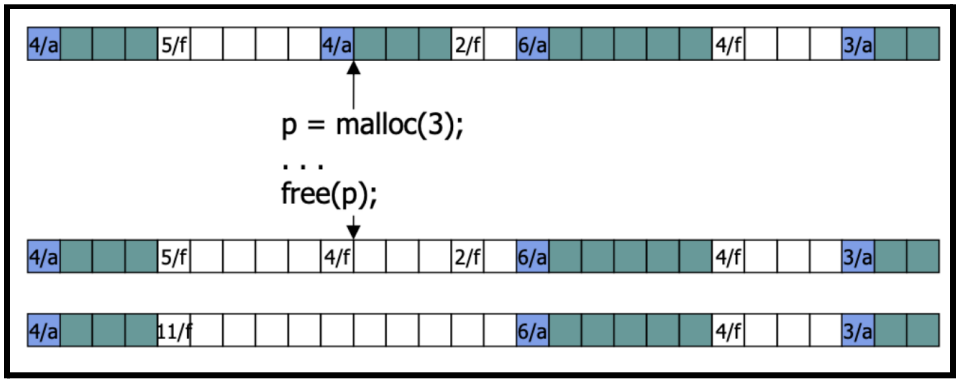
Lecture Notes:

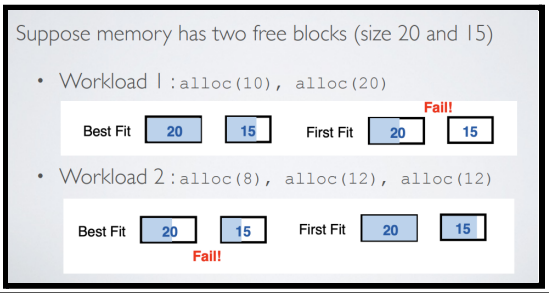
- **Managing Free Memory:**
- There are 2 types of memory allocation:
    1. **Static Allocation/Stack Allocation:**
    - Fixed in size.
    - Uses data structures that do not need to grow or shrink such as global and local variables.
      E.g. **char name[16];**
    - Done at compile time.
    - Restricted, but simple and efficient.
    2. **Dynamic Allocation/Heap Allocation:**
    - Changes in size.
    - Uses data structures that might increase/decrease in size according to different demands.
      E.g. **name = (char *) malloc(16);**
    - Done at run time.
    - General, but difficult to implement.
- Heap allocation is used to manage contiguous ranges of logical addresses.
- **malloc(size)** returns a pointer to a block of memory of at least size bytes, or NULL.
- **free(ptr)** releases the previously- allocated block pointed to by ptr.
- Heap allocation is difficult because using free() creates a lot of holes (**fragmentation**).
- Fragmentation is the inability to use memory that is free.
- Two factors are required for fragmentation:
    1. **Different lifetimes:** If all objects die at the same time, then there is no fragmentation.
    2. **Different sizes:** If all requests are the same size, then there is no fragmentation.
- Some important decisions:
    1. **Placement choice: Where in free memory to put a requested block?**
    - Freedom: Can select any memory in the heap.
    - Ideal: Put the block where it won't cause fragmentation later. This is impossible in general because it requires future knowledge.
    2. **Split free blocks to satisfy smaller requests?**
    - Freedom: Can choose any larger block to split.
    - Ideal: Choose specific blocks to minimize fragmentation.
- **Note:** Fragmentation is impossible to solve.
- Theoretical result: For any allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation L.
- **Heap Memory Allocator:**
- What the memory allocator must do:
    - Track which parts of memory are in use, and which parts are free. Ideally, there should be no wasted space and no time overhead.
- What the memory allocator cannot do:
    - Control the order of the number and size of requested blocks.
    - Know the number, size, & lifetime of future allocations.
- What makes a good memory allocator:
    - The one that avoids compaction (time consuming).
    - The one that minimizes fragmentation.
- **Tracking memory allocation with bitmaps:**
- **Bitmap:** 1 bit per allocation unit.
  0 means free
  1 means allocated

- Allocating a N-unit chunk requires scanning a bitmap for a sequence of N zero's.
- This process is very slow.
- **Tracking memory allocation with lists:**
- The **free lists** maintain a linked list of allocated and free segments.
- In an **implicit list**, each block has a header that records size and status (allocated or free). Searching for free blocks is linear in total number of blocks.
- An **explicit list** stores pointers in free blocks to create a doubly-linked list.
- **Freeing blocks:**
- Adjacent free blocks can be coalesced (merged).
  E.g.



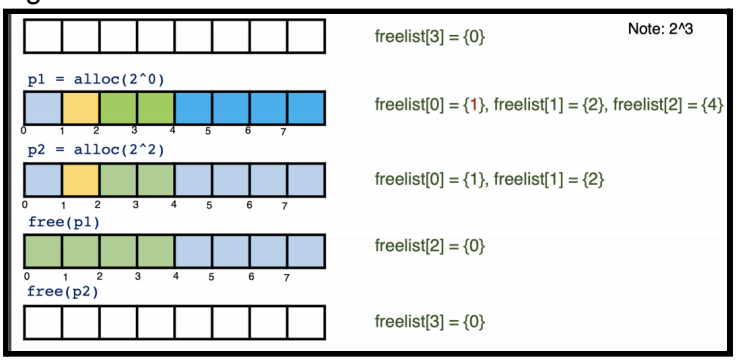
- **Placement Algorithms:**
- There are 5 placement algorithms that can be used for merging free blocks:
  1. **First-fit:** Choose the first block that is large enough. The search can start at the beginning, or where the previous search ended.
  2. **Best-fit:** Choose the block that is closest in size to the request.
  3. **Worst-fit:** Choose the largest block.
  4. **Quick-fit:** Keep multiple free lists for common block sizes.
  5. **Buddy systems:** Round up allocations to power of 2 to make management faster.
- **Best Fit:**
- Minimizes fragmentation by allocating space from blocks that leave the smallest fragment.
- The best data structure to use is a heap as it is a list of free blocks where each has a header holding block size and a pointer to the next block.
- The idea is to search freelist for the block closest in size to the request.
- **First Fit:**
- Pick the first block that fits.
- Data structures that can be used for this include free list, sorted LIFO, FIFO, or by address.
- The idea is to scan the list and take the first one.

- **Best Fit vs First Fit:**



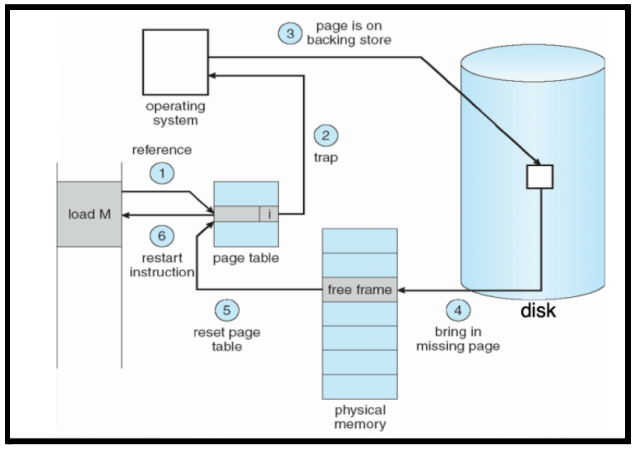
	First Fit	Best Fit
Advantage	Simplest, and often fastest and most efficient	In practice, similar storage utilization to first-fit
Disadvantage	May leave many small fragments near start of memory that must be searched repeatedly	Left-over fragments tend to be small (unusable)

- **Buddy Allocation:**
- Allocate blocks in 2^k.
- For the data structure, maintain n free lists of blocks of size 2^0, 2^1, …, 2^n.
- The idea is this:
  - Recursively divide larger blocks until they reach a suitable block.
  - Insert buddy blocks into free lists.
  - Upon free, recursively coalesce block with buddy if buddy is free.
    Note: The addresses of the buddy pair only differ by one bit.
- Linux uses this approach.
- E.g.



- Advantages:
  - Fast search (allocate) and merge (free).
  - Avoid iterating through the free list.
  - Avoid external fragmentation for req of 2^n.
  - Keep physical pages contiguous.

- **Page Replacements Algorithms:**
- Swapping is when we use a disk to simulate a larger virtual than physical memory. I.e.

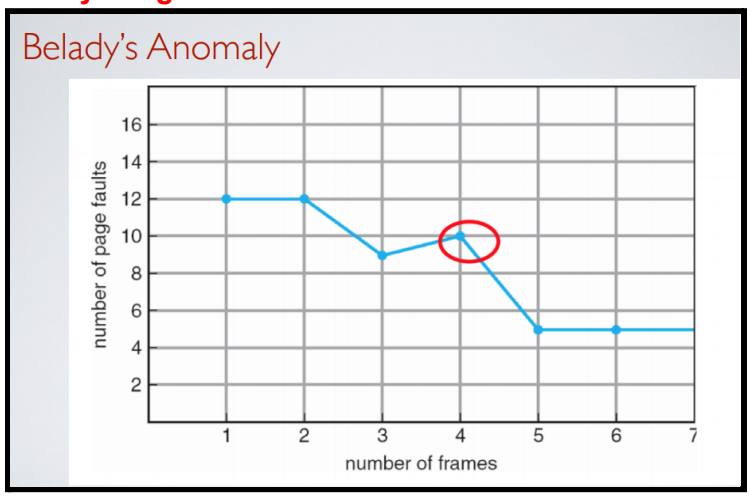


- What happens when there is a page fault? ➡ The OS loads the faulted page frame from disk into physical memory.
- What when there is no physical memory available or the process has reached its limit of maximum page frame allowed? ➡ The OS must evict an existing frame to replace it with the new one.
- How to determine which page frame should be evicted? ➡ The **page replacement algorithm**/**page eviction policy** determines which page frame to evict to minimize the fault rate (affecting paging performances).
- **Page Replacement Algorithms:**
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.
- There are 3 main algorithms:
    1. **FIFO (First In, First Out):** Evict the oldest page in the system.
    2. **LRU (Last Recently Used):** Evict the page that has not been used for the longest time in the past.
    3. **Second Chance:** An approximation of LRU that is more implementable.
- Replacement algorithms are evaluated on a reference string by counting the number of page faults.
- **FIFO:**
- Evict the oldest page in the system.
- We will only have 3 physical pages in this example.

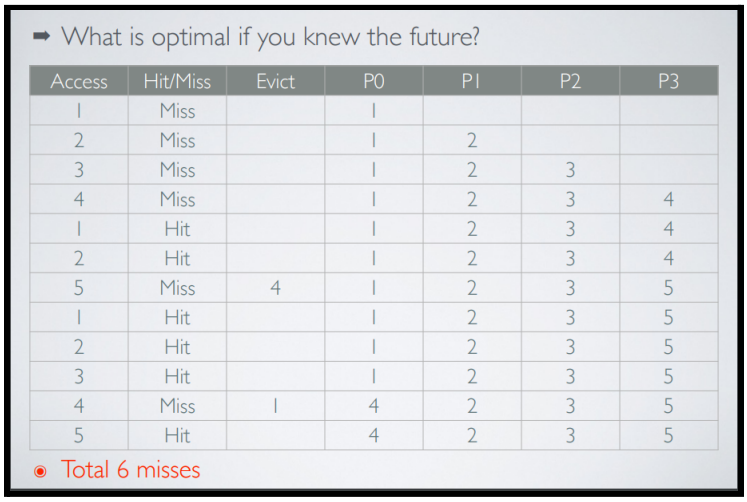
Access	Hit/Miss	Evict	P0	P1	P2
1	Miss		1		
2	Miss		1	2	
3	Miss		1	2	3
4	Miss	1	4	2	3
1	Miss	2	4	1	3
2	Miss	3	4	1	2
5	Miss	4	5	1	2
1	Hit		5	1	2
2	Hit		5	1	2
3	Miss	1	5	3	2
4	Miss	2	5	3	4
5	Hit		5	3	4

- Total 9 misses

- Does having more physical memory automatically means fewer page faults? The answer is no, more physical memory doesn't always mean fewer faults. This is proven by **Belady's Algorithm**.



- Belady tried to find the most optimal number of page frames if you could see the future, shown below.



Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	4	1	2	3	5
1	Hit		1	2	3	5
2	Hit		1	2	3	5
3	Hit		1	2	3	5
4	Miss	1	4	2	3	5
5	Hit		4	2	3	5

- Belady's Algorithm is known and proven to be the optimal page replacement algorithm. The problem is that it is hard (nearly impossible) to predict the future.
  Belady's algorithm is useful to compare page replacement algorithms with the optimal to gauge room for improvement.

- **LRU:**
- Evict the page that has not been used for the longest time in the past.
- We will only have 3 physical pages in this example.

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	3	1	2	5	4
1	Hit		1	2	5	4
2	Hit		1	2	5	4
3	Miss	4	1	2	5	3
4	Miss	5	1	2	4	3
5	Miss	1	5	2	4	3

- Total 8 misses

- There are 2 ways to implement LRU:
    1. Stamp the pages with timer value.
       On access, stamp the PTE with the timer value.
       On a miss, scan the page table to find the oldest counter value.
       The problem is that this would double memory traffic.
    2. Use a doubly-linked list of pages.
       On access, move the page to the tail.
       On a miss, remove the head page.
       The problem with this is that, again, it is very expensive.
- So, we need to approximate LRU instead. This is where the Second Chance page replacement algorithm comes in.
- **Second Chance:**
- We will only have 3 physical pages in this example.

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1*	2	3	4
2	Hit		1*	2*	3	4
5	Miss	3	1	2	5	4
1	Hit		1*	2	5	4
2	Hit		1*	2*	5	4
3	Miss	4	1*	2*	5	3
4	Miss	5	1	2	4	3
5	Miss	3	1	2	4	5

- Total 8 misses

- There are 2 ways to implement second chance:
    1. FIFO-like algorithm:
        - Use the accessed bit supported by most hardware.
        - Data structure: A linked list of pages with two pointers head and tail.
        - Code:
            - On hit, set the corresponding page's accessed bit to 1.
            - On miss:
                1. While the head's accessed bit is 1, set head's accessed bit to 0 and move it to tail.
                2. Otherwise, the head's accessed bit is 0, swap the head and move the new page to tail.
        - Good performances but requires moving pages on every miss.
    2. Clock algorithm:
        - Use the accessed bit supported by most hardware.
        - Data structure: A circular linked list of pages (clock) with 1 pointer (hand).
        - Code:
            - On hit, set the corresponding page's accessed bit to 1.
            - On miss:
                1. While the hand's accessed bit is 1, set the hand's accessed bit to 0 and move to the next page.
                2. Otherwise, if the hand's accessed bit is 0, swap the hand's page with the new page and move to the next page.
        - Better performances than fifo-like second chance (no rotation on miss)
- Some other replacement algorithms include:
    - **Random eviction:**
        - Very simple to implement.
        - Not overly horrible (avoids Belady's anomaly).
    - **LFU (least frequently used) eviction:**
        - Instead of just a bit, count the number of times each page was accessed. The least frequently accessed must not be very useful or maybe was just brought in and is about to be used.
        - Decay usage counts over time for pages that fall out of usage.
    - **MFU (most frequently used) algorithm:**
        - Because the page with the smallest count was probably just brought in and has yet to be used, it will be needed in the future.
- Neither LFU nor MFU are used very commonly.
- **Working Set Model:**
- How can we determine how much memory to give to each process?
- **Fixed space algorithms:**
    - Each process is given a limit of pages it can use.
    - When it reaches the limit, it replaces from its own pages.
    - Local replacement : Some processes may do well while others suffer.
- **Variable space algorithms:**
    - Process' set of pages grows and shrinks dynamically.
    - Global replacement: One process can ruin it for the rest.
- A **working set (WS)** of a process is used to model the dynamic locality of its memory usage.
- WS(t,w) = {pages P | P was referenced in the time interval (t, t-w)}
  t – time, w – working set window (measured in page refs)
- A page is in the working set only if it was referenced in the last w references.

- The **working set size** is the number of unique pages in the working set.
  I.e. It is the number of pages referenced in the interval (t, t-w).
- The working set size changes with program locality.
  During periods of poor locality, you reference more pages.
  Within that period of time, the working set size is larger.
- Intuitively, we want the working set to be the set of pages a process needs in memory to prevent heavy faulting.
  Each process has a parameter w that determines a working set with few faults.
  Don't run a process unless the working set is in memory.
- Some problems for the working set:
  1. Hard to determine w.
  2. Hard to know when the working set changes.
     - However, it is still used as an abstraction.
       For example, when people ask, "How much memory does Firefox need?", they are in effect asking for the size of Firefox's working set.
- **Page Fault Frequency (PFF):**
- **Page Fault Frequency (PFF)** is a variable space algorithm that uses a more ad-hoc approach.
- Monitor the fault rate for each process:
  - If the fault rate is above a high threshold, give it more memory.
  - If the fault rate is below a low threshold, take away memory.
- It is hard to use PFF to distinguish between changes in locality and changes in size of the working set.
- **Thrashing:**
- An **overcommitted system** occurs when an OS spends most of the time paging data back and forth from the disk and spending little time doing useful work.
- The problem comes from either:
  - A bad page replacement algorithm (that does not help minimizing page fault) OR
  - There is not enough physical memory for all processes.
- **Windows XP Paging Policy:**
- Local page replacement.
- Per-process FIFO.
- Processes start with a default of 50 pages.
- XP monitors page fault rate and adjusts working-set size accordingly.
- On page faults, clusters of pages around the missing page are brought into memory.
- **Linux Paging:**
- Global replacement (like most Unix).
- Modified second-chance clock algorithm.
- Pages age with each pass of the clock hand.
- Pages that are not used for a long time will eventually have a value of zero.

Textbook Notes:

- **Beyond Physical Memory - Mechanisms:**
- Introduction:
- Thus far, we've assumed that an address space is unrealistically small and fits into physical memory. In fact, we've been assuming that every address space of every running process fits into memory. We will now relax these big assumptions, and assume that we wish to support many concurrently-running large address spaces.
- To do so, we require an additional level in the memory hierarchy. Thus far, we have assumed that all pages reside in physical memory. However, to support large address spaces, the OS will need a place to stash away portions of address spaces that currently aren't in great demand. In general, the characteristics of such a location are that it

should have more capacity than memory; as a result, it is generally slower. In modern systems, this role is usually served by a hard disk drive. Thus, in our memory hierarchy, big and slow hard drives sit at the bottom, with memory just above. This is for convenience and ease of use.

- With a large address space, you don't have to worry about if there is room enough in memory for your program's data structures; rather, you just write the program naturally, allocating memory as needed. It is a powerful illusion that the OS provides, and makes your life vastly simpler. A contrast is found in older systems that used **memory overlays**, which required programmers to manually move pieces of code or data in and out of memory as they were needed.
- Beyond just a single process, the addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently running processes. The invention of multiprogramming almost demanded the ability to swap out some pages, as early machines clearly could not hold all the pages needed by all processes at once. Thus, the combination of multiprogramming and ease-of-use leads us to want to support using more memory than is physically available. It is something that all modern VM systems do.
- Swap Space:
- The first thing we will need to do is to reserve some space on the disk for moving pages back and forth. In operating systems, we generally refer to such space as **swap space**, because we swap pages out of memory to it and swap pages into memory from it. Thus, we will simply assume that the OS can read from and write to the swap space, in page-sized units. To do so, the OS will need to remember the **disk address** of a given page.
- The size of the swap space is important, as ultimately it determines the maximum number of memory pages that can be in use by a system at a given time.
- We should note that swap space is not the only on-disk location for swapping traffic. For example, assume you are running a program binary. The code pages from this binary are initially found on disk, and when the program runs, they are loaded into memory (either all at once when the program starts execution, or, as in modern systems, one page at a time when needed). However, if the system needs to make room in physical memory for other needs, it can safely re-use the memory space for these code pages, knowing that it can later swap them in again from the on-disk binary in the file system.
- The Present Bit:
- Let us assume, for simplicity, that we have a system with a hardware-managed TLB.
- Recall first what happens on a memory reference. The running process generates virtual memory references (for instruction fetches, or data accesses), and, in this case, the hardware translates them into physical addresses before fetching the desired data from memory.
- Remember that the hardware first extracts the VPN from the virtual address, checks the TLB for a match (a TLB hit), and if a hit, produces the resulting physical address and fetches it from memory. This is hopefully the common case, as it is fast (requiring no additional memory accesses). If the VPN is not found in the TLB (a TLB miss), the hardware locates the page table in memory using the page table base register and looks up the page table entry (PTE) for this page using the VPN as an index. If the page is valid and present in physical memory, the hardware extracts the PFN from the PTE, installs it in the TLB, and retries the instruction, this time generating a TLB hit.
- If we wish to allow pages to be swapped to disk, however, we must add even more machinery. Specifically, when the hardware looks in the PTE, it may find that the page is not present in physical memory. The way the hardware or OS determines this is through a new piece of information in each page-table entry, known as the **present bit**. If the

present bit is set to one, it means the page is present in physical memory and everything proceeds as above; if it is set to zero, the page is not in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

- Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a **page-fault handler**, runs, and must service the page fault.
- The Page Fault:
- Recall that with TLB misses, we have two types of systems: hardware-managed TLBs (where the hardware looks in the page table to find the desired translation) and software-managed TLBs (what the OS does). In either type of system, if a page is not present, the OS is put in charge to handle the page fault. The **OS page-fault handler** runs to determine what to do. Virtually all systems handle page faults in software. Even with a hardware-managed TLB, the OS manages this important duty.
- If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.
- When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation (one could alternately update the TLB when servicing the page fault to avoid this step). Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address.
- Note that while the I/O is in flight, the process will be in the blocked state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware.
- What If Memory Is Full:
- In the process described above, you may notice that we assumed there is plenty of free memory in which to page in a page from swap space. Of course, this may not be the case. Thus, the OS might like to first page out one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out, or replace is known as the **page-replacement policy**.
- As it turns out, a lot of thought has been put into creating a good page replacement policy, as kicking out the wrong page can exact a great cost on program performance. Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds; in current technology that means a program could run 10,000 or 100,000 times slower.
- Page Fault Control Flow:
- There are now three important cases to understand when a TLB miss occurs.
- First, that the page was both present and valid. In this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction this time resulting in a TLB hit, and thus continue as described before.
- In the second case, the page fault handler must be run. Although this was a legitimate page for the process to access it is not present in physical memory.

- Third, the access could be to an invalid page, due for example to a bug in the program. In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.
- This is what the OS roughly must do in order to service the page fault. First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we'll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here. With a physical frame in hand, the handler then issues the I/O request to read in the page from swap space. Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another retry, a TLB hit, at which point the hardware will be able to access the desired item.
- When Replacements Really Occur:
- Thus far, the way we've described how replacements occur assumes that the OS waits until memory is entirely full, and only then replaces a page to make room for some other page. As you can imagine, this is a little bit unrealistic, and there are many reasons for the OS to keep a small portion of memory free more proactively.
- To keep a small amount of memory free, most operating systems thus have some kind of high watermark (HW) and low watermark (LW) to help decide when to start evicting pages from memory. How this works is as follows: when the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available. The background thread, sometimes called the **swap daemon** or **page daemon**, then goes to sleep, happy that it has freed some memory for running processes and the OS to use.
- Beyond Physical Memory - Policies:
- In a virtual memory manager, life is easy when you have a lot of free memory. A page fault occurs, you find a free page on the free-page list, and assign it to the faulting page. Unfortunately, things get a little more interesting when little memory is free. In such a case, this memory pressure forces the OS to start paging out pages to make room for actively-used pages. Deciding which page(s) to evict is encapsulated within the replacement policy of the OS. Historically, it was one of the most important decisions the early virtual memory systems made, as older systems had little physical memory.
- Cache Management:
- Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a cache for virtual memory pages in the system. Thus, our goal in picking a replacement policy for this cache is to minimize the number of cache misses, i.e., to minimize the number of times that we have to fetch a page from disk. Alternatively, one can view our goal as maximizing the number of cache hits, i.e., the number of times a page that is accessed is found in memory.
- Knowing the number of cache hits and misses let us calculate the **average memory access time (AMAT)** for a program.

$$AMAT = T_M + (P_{Miss} \cdot T_D)$$

T_M represents the cost of accessing memory, T_D the cost of accessing disk, and P_{Miss} the probability of not finding the data in the cache. P_{Miss} varies from 0.0 to 1.0, and sometimes we refer to a percent miss rate instead of a probability (E.g. A 10% miss rate means PMiss = 0.10).

Note: You always pay the cost of accessing the data in memory. When you miss, however, you must additionally pay the cost of fetching the data from disk.

- The Optimal Replacement Policy:
- The logic is like this: If you have to throw out some page, why not throw out the one that is needed the furthest from now? By doing so, you are essentially saying that all the other pages in the cache are more important than the one furthest out. The reason this is true is simple: you will refer to the other pages before you refer to the one furthest out.
- In the computer architecture world, architects sometimes find it useful to characterize misses by type, into one of three categories: **compulsory**, **capacity**, and **conflict misses**, sometimes called the Three C's.
- A **compulsory miss** or **cold-start miss** occurs because the cache is empty to begin with and this is the first reference to the item.
- A **capacity miss** occurs because the cache ran out of space and had to evict an item to bring a new item into the cache.
- A **conflict miss** arises in hardware because of limits on where an item can be placed in a hardware cache, due to **set associativity**. It does not arise in the OS page cache because such caches are always fully-associative. I.e. There are no restrictions on where in memory a page can be placed.
- A Simple Policy - FIFO:
- Many early systems avoided the complexity of trying to approach optimal and employed very simple replacement policies.
- For example, some systems used **FIFO replacement**, where pages were simply placed in a queue when they enter the system. When a replacement occurs, the page on the tail of the queue is evicted. FIFO has one great strength: it is quite simple to implement.
- Another Simple Policy - Random:
- Another similar replacement policy is Random, which simply picks a random page to replace under memory pressure. Random has properties similar to FIFO; it is simple to implement, but it doesn't really try to be too intelligent in picking which blocks to evict.
- Using History - LRU:
- Unfortunately, any policy as simple as FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again. FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon be paged back in.
- As we did with scheduling policy, to improve our guess at the future, we once again lean on the past and use history as our guide.
- One type of historical information a page-replacement policy could use is **frequency**; if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value.
- A more commonly used property of a page is its **recency of access**; the more recently a page has been accessed, perhaps the more likely it will be accessed again.
- This family of policies is based on what people refer to as the **principle of locality**, which basically is just an observation about programs and their behavior. What this principle says, quite simply, is that programs tend to access certain code sequences (e.g. in a loop) and data structures (e.g. an array accessed by the loop) quite frequently; we should thus try to use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.
- And thus, a family of simple historically-based algorithms are born. The **Least-Frequently-Used (LFU)** policy replaces the least-frequently used page when an eviction must take place. Similarly, the **Least-Recently-Used (LRU)** policy replaces the least-recently-used page.

- There are two types of locality that programs tend to exhibit. The first is known as **spatial locality**, which states that if a page P is accessed, it is likely the pages around it (say P − 1 or P + 1) will also likely be accessed. The second is **temporal locality**, which states that pages that have been accessed in the near past are likely to be accessed again in the near future. The assumption of the presence of these types of locality plays a large role in the caching hierarchies of hardware systems, which deploy many levels of instruction, data, and address-translation caching to help programs run fast when such locality exists.
- Of course, the **principle of locality**, as it is often called, states that there is no hard-and-fast rule that all programs must obey. Indeed, some programs access memory or disk in rather random fashion and don't exhibit much or any locality in their access streams. Thus, while locality is a good thing to keep in mind while designing caches of any kind (hardware or software), it does not guarantee success. Rather, it is a heuristic that often proves useful in the design of computer systems.
- Approximating LRU:
- The idea requires some hardware support, in the form of a **use bit**, sometimes called the **reference bit**, the first of which was implemented in the first system with paging.
- There is one use bit per page of the system, and the use bits live in memory somewhere. Whenever a page is referenced (read or written), the use bit is set by hardware to 1. The hardware never clears the bit. That is the responsibility of the OS.
- How does the OS employ the use bit to approximate LRU? Well, there could be a lot of ways, but with the clock algorithm, one simple approach was suggested. Imagine all the pages of the system arranged in a circular list. A clock hand points to some particular page to begin with (it doesn't really matter which). When a replacement must occur, the OS checks if the currently-pointed page P has a use bit of 1 or 0. If 1, this implies that page P was recently used and thus is not a good candidate for replacement. Thus, the use bit for P is set to 0 (cleared), and the clock hand is incremented to the next page (P + 1). The algorithm continues until it finds a use bit that is set to 0, implying this page has not been recently used (or, in the worst case, that all pages have been and that we have now searched through the entire set of pages, clearing all the bits).
- Note that this approach is not the only way to employ a use bit to approximate LRU. Indeed, any approach which periodically clears the use bits and then differentiates between which pages have use bits of 1 versus 0 to decide which to replace would be fine.
- Considering Dirty Pages:
- One small modification to the clock algorithm that is commonly made is the additional consideration of whether a page has been modified or not while in memory. The reason for this: if a page has been modified and is thus dirty, it must be written back to disk to evict it, which is expensive. If it has not been modified and is thus clean, the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages.
- To support this behavior, the hardware should include a **modified bit**/**dirty bit**. This bit is set any time a page is written, and thus can be incorporated into the page-replacement algorithm. The clock algorithm, for example, could be changed to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.
- Other VM Policies:
- Page replacement is not the only policy the VM subsystem employs though it may be the most important. For example, the OS also has to decide when to bring a page into memory. This policy, sometimes called the **page selection policy**, presents the OS with some different options.

- For most pages, the OS simply uses **demand paging**, which means the OS brings the page into memory when it is accessed, “on demand” as it were. Of course, the OS could guess that a page is about to be used, and thus bring it in ahead of time; this behavior is known as **prefetching** and should only be done when there is a reasonable chance of success. For example, some systems will assume that if a code page P is brought into memory, that code page P + 1 will likely soon be accessed and thus should be brought into memory too.
- Another policy determines how the OS writes pages out to disk. Of course, they could simply be written out one at a time; however, many systems instead collect a number of pending writes together in memory and write them to disk in one (more efficient) write. This behavior is usually called **clustering** or simply grouping of writes, and is effective because of the nature of disk drives, which perform a single large write more efficiently than many small ones.
- Thrashing:
- Before closing, we address one final question: what should the OS do when memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory? In this case, the system will constantly be paging, a condition sometimes referred to as **thrashing**.
- Some earlier operating systems had a fairly sophisticated set of mechanisms to both detect and cope with thrashing when it took place. For example, given a set of processes, a system could decide not to run a subset of processes, with the hope that the reduced set of processes’ working sets (the pages that they are using actively) fit in memory and thus can make progress. This approach, generally known as **admission control**, states that it is sometimes better to do less work well than to try to do everything at once poorly, a situation we often encounter in real life as well as in modern computer systems.
- Some current systems take a more draconian approach to memory overload. For example, some versions of Linux run an out-of-memory killer when memory is oversubscribed; this daemon chooses a memory intensive process and kills it, thus reducing memory in a none-too-subtle manner. While successful at reducing memory pressure, this approach can have problems, if, for example, it kills the X server and thus renders any applications requiring the display unusable.